Title: Stack and Heap Memory in Java

---

1. Introduction

In Java, memory management is divided mainly into **Stack** and **Heap** memory. Understanding how they work is crucial for efficient coding, object management, and avoiding errors like `StackOverflowError` or memory leaks.

---

2. Stack Memory

2.1 What is Stack Memory?

- Stores **primitive local variables** and **references to objects**.
- Works on **LIFO (Last In, First Out)** principle.
- Each method call creates a **stack frame**.
- Automatically removed when method execution ends.

2.2 Example

```
class Demo {
    void methodA() {
        int x = 10;
        int y = 20;
        methodB();
    }

    void methodB() {
        int z = 30;
    }

    public static void main(String[] args) {
        Demo obj = new Demo();
        obj.methodA();
    }
}
```

2.3 Stack Flow

1. `main()` called → stack frame for main created (variable: `obj`)
2. `methodA()` called → new stack frame added (`x=10, y=20`)
3. `methodB()` called → new stack frame added (`z=30`)
4. `methodB()` finishes → stack frame removed
5. `methodA()` finishes → stack frame removed
6. `main()` finishes → stack empty

2.4 Rules of Stack

- LIFO principle.
- Local variables exist only during method execution.
- Too many method calls → `StackOverflowError`.

---

3. Heap Memory

3.1 What is Heap Memory?

- Stores **objects and arrays**.
- Shared among all threads.
- Managed by **Garbage Collector (GC)**.
- Objects live **as long as references exist**.

3.2 Example

```
class Demo {
    int data;
    Demo(int d) { data = d; }
}

public class Main {
    public static void main(String[] args) {
        Demo obj1 = new Demo(10); // reference in stack, object in heap
        Demo obj2 = new Demo(20); // reference in stack, object in heap
        Demo obj3 = obj1;         // same object as obj1 in heap
    }
}
```

3.3 Rules of Heap

- Stores all objects and arrays.
- Objects live as long as references exist.
- GC automatically removes unreferenced objects.
- Memory allocation is dynamic.
- Accessible by multiple threads.

---

4. Garbage Collector (GC)

- Automatically frees memory in heap for unreferenced objects.
- Prevents memory leaks.
- Runs in background; exact timing cannot be predicted.

Example:

```
Demo obj = new Demo(10);
obj = null; // object eligible for GC
```

---

5. Stack vs Heap Comparison

Feature	Stack	Heap
Stores	Primitive variables, references	Objects, arrays
Access speed	Fast	Slower
Size	Limited	Larger, tunable
Lifetime	Short-lived (method execution)	Long-lived (as long as referenced)
Management	LIFO, automatic	Garbage Collector
Shared among threads	No	Yes

---

6. Visualization

Program Example:

```
class Demo {
    int data;
    Demo(int d) { data = d; }
}

public class Main {
    public static void main(String[] args) {
        Demo obj1 = new Demo(10);
        Demo obj2 = new Demo(20);
        int x = 5;
        int y = 15;
    }
}
```

Stack (during main execution):

```
+-----------------+
| main() frame     |
| x = 5           |
| y = 15          |
| obj1 -> ref1    |
```

```
| obj2 -> ref2    |
+-----------------+
```

Heap:

```
+---------+        +---------+
| Object1 | <------| obj1 ref|
| data=10 |        +---------+
+---------+

+---------+        +---------+
| Object2 | <------| obj2 ref|
| data=20 |        +---------+
+---------+
```

- Stack holds **primitives + references**
- Heap holds **actual objects**
- Once `obj1` or `obj2` references are null → GC can remove them from heap

---

7. Key Takeaways

1. Stack → small, fast, stores primitives & references, LIFO.
2. Heap → large, slower, stores objects & arrays, GC-managed.
3. Local variables exist in stack; objects exist in heap.
4. Understanding stack vs heap is crucial for **memory management, performance, and avoiding errors**.